Learning union of k-testable languages

Statistical and symbolic language modeling project

Rania el Bouhssini, Martin Laville and Félix Jamet

December 25, 2018

Contents

1 Introduction

Unless explicitely specified, all definitions and algorithms in this document are coming from Linard *et al.* (2018), which we will sometimes refer to as "the paper".

We will present a possible implementation of those definitions and algorithms in a modular fashion, using Python3. Modular meaning here that we will implement concepts as they come and assemble them later as a whole when the necessary parts are complete. So if an `__init__` appears in the wild without its enclosing `class`, it's nothing to worry about.

2 k-testable languages

A k-testable language is a language that can be recognised by sliding a window of size k over an input. By definition, we have $k > 0$ since sliding a window of null or negative size would hardly make any sense.

A language is said to be k-testable in the strict sense (k-TSS) if it can be represented using a construct called a k-test vector. The necessary informations and operations on this construct will be implemented in the `ktestable` class.

2.1 k-test vector

A k-test vector is a 4-tuple $Z = \langle I, F, T, C \rangle$:

— $I \in \Sigma^{k-1}$ is a set of allowed prefixes,
— $F \in \Sigma^{k-1}$ is a set of allowed suffixes,
— $T \in \Sigma^{k}$ is a set of allowed segments, and
— $C \in \Sigma^{<k}$ is a set of allowed short strings satisfying $I \cap F = C \cap \Sigma^{k-1}$.

We will refer to I, F, T and C respectively as the allowed prefixes, suffixes, infixes and short strings. Moreover, we will refer to $I \cap F$, *i.e.* the prefixes that are also suffixes as presuffixes. An intuitive way to formulate the constraint on short strings is that the short strings of length $k-1$ have to be presuffixes and vice versa.

This definition can be translated into an init.

Init k-test vector:

```
def __init__(self, prefixes, suffixes, infixes, shorts, k=None):
    self.k = len(next(iter(infixes))) if k is None else k
    self.prefixes = prefixes
    self.suffixes = suffixes
    self.infixes = infixes
    self.shorts = shorts
    self.ensure_correct_definition()
```

We then write `ensure_correct_definition` to make sure that the created k-test vector respects the conditions of the definition.

Ensure correct definition:

```
def ensure_correct_definition(self):
    def same_length(collection, reference_length):
        return all(map(lambda x: len(x) == reference_length, collection))

    errors = []
    if not same_length(self.prefixes, self.k - 1):
        errors.append('incorrect prefix length')
    if not same_length(self.suffixes, self.k - 1):
        errors.append('incorrect suffix length')
    if not same_length(self.infixes, self.k):
        errors.append('incorrect infix length')
    if not all(map(lambda x: len(x) < self.k, self.shorts)):
        errors.append('incorrect short string length')

    presuffixes = self.prefixes & self.suffixes
    shorts_len_k = set(filter(lambda x: len(x) == self.k - 1, self.shorts))
    if presuffixes != shorts_len_k:
        errors.append('short strings conditions not satisfied')

    if len(errors) >0:
        raise ValueError(', '.join(errors).capitalize() + '.')
```

2.2 k-test vectors as a partially ordered set

Let $\mathcal{T}_k$ be the set of all k-test vectors. A partial order $\sqsubseteq$ can be defined on $\mathcal{T}_k$ as follow:

$$\langle I, F, T, C \rangle \sqsubseteq \langle I', F', T', C' \rangle \iff I \subseteq I' \wedge F \subseteq F' \wedge T \subseteq T' \wedge C \subseteq C'$$

With this partial order, a union, an intersection and a symmetric difference can be defined on the k-test vectors $Z = \langle I, F, T, C \rangle$ and $Z' = \langle I', F', T', C' \rangle$.

First, we need to be able to check wether two `testable` are compatible, *i.e.* wether they have the same `k`.

k-test vector compatibility:

```
def ensure_compatibility(self, other):
    if self.k != other.k:
        raise ValueError('Incompatible k-test vectors: length mismatch (%d != %d)' %
                         (self.k, other.k))
```

2.2.1 Union ($\sqcup$)

$$Z \sqcup Z' = \langle I \cup I', F \cup F', T \cup T', C \cup C' \cup (I \cap F') \cup (I' \cap F) \rangle$$

We can see that the constraint on short strings $I \cap F = C \cap \Sigma^{k-1}$ is still respected because the short strings are updated with all the cases which could contradict it.

The implementation is a quite litteral translation if this definition.

k-test vector union:

```
def union(self, other):
    self.ensure_compatibility(other)
    prefixes = self.prefixes | other.prefixes
    suffixes = self.suffixes | other.suffixes
    infixes = self.infixes | other.infixes
    shorts = self.shorts | other.shorts |\
        (self.prefixes & other.suffixes) |\
        (self.suffixes & other.prefixes)
    return ktestable(prefixes, suffixes, infixes, shorts, k=self.k)
```

2.2.2 Intersection ($\sqcap$)

$Z \sqcap Z' = \langle I \cap I', F \cap F', T \cap T', C \cap C' \rangle$

Once again, the implementation is straightforward.

k-test vector intersection:

```
def intersection(self, other):
    self.ensure_compatibility(other)
    prefixes = self.prefixes & other.prefixes
    suffixes = self.suffixes & other.suffixes
    infixes = self.infixes & other.infixes
    shorts = self.shorts & other.shorts
    return ktestable(prefixes, suffixes, infixes, shorts, k=self.k)
```

2.2.3 Symmetric difference ($\triangle$)

$Z \triangle Z' = \langle I \triangle I', F \triangle F', T \triangle T', C \triangle C' \triangle (I \cap F') \triangle (I' \cap F) \rangle$

Once more, it's only a matter of translating the set operations into python code.

k-test vector symmetric difference:

```
def symmetric_difference(self, other):
    self.ensure_compatibility(other)
    prefixes = self.prefixes ^ other.prefixes
    suffixes = self.suffixes ^ other.suffixes
    infixes = self.infixes ^ other.infixes
    shorts = self.shorts ^ other.shorts ^\
        (self.prefixes & other.suffixes) ^\
        (self.suffixes & other.prefixes)
    return ktestable(prefixes, suffixes, infixes, shorts, k=self.k)
```

2.2.4 Operators

Since the semantic of the three operations defined above are similar to those of sets, we create operators for them, matching the operators of `set`, the python builtin. That is to say `|` for union, `&` for intersection and `^` for symmetric difference.

k-test vector operators:

```
def __or__(self, other):
    return self.union(other)

def __and__(self, other):
    return self.intersection(other)

def __xor__(self, other):
    return self.symmetric_difference(other)
```

2.3 Measures

The theorem 3 of Linard *et al.* (2018) states that

> Any language that is a union of k-TSS languages can be identified in the limit from positive examples.

We will call "a union of k-TSS languages" a k-TSS-union. This Theorem means that when trying to learn a k-TSS-union from examples, the language will be learned at some point, having only seen a finite number of examples, even though the language might have an infinite number of examples.

It provides us with a baseline algorithm to learn a k-TSS-union. We consider each example as a language of its own and take the union of those examples. One problem of this algorithm is that it requires a great number of k-test vectors and will thus tend to be computationnaly expensive.

The solution to this problem is to consider it as a clustering problem by putting together similar vectors. The clustering algorithm will be seen later. Before this, there is a need to define a metric on k-test vectors, metric which will use the notion of cardinality.

2.3.1 Cardinality

The cardinality of a k-test vector $Z = \langle I, F, T, C \rangle$ is defined as:

$$|Z| = |I| + |F| + |T| + |C \cap \Sigma^{k-1}|$$

Once again, we see the influence of the short strings constraint since only the short strings of length less then $k-1$ are taken into account. Curiously, there is nothing in place to compensate for the presuffixes being counted twice. An alternative measure that takes this deduplication into account can be defined as:

$$|Z| = |I| + |F| + |T| + |C \cap \Sigma^{k-1}| - |I \cap F|$$

But we will still use the original definition.

k-test vector cardinality:

```
def cardinality(self):
    return len(self.prefixes) + len(self.suffixes) + len(self.infixes) +\
        sum(map(lambda x: 1 if len(x) < self.k - 1 else 0, self.shorts))

def __len__(self):
    return self.cardinality()
```

We also defined the operator `len`, since the meaning is similar to the builtin `len` of python sets.

2.3.2 Distance

The distance between two k-test vectors is the cardinality of their symmetric difference:

$$d(Z, Z') = |Z \triangle Z'|$$

It corresponds intuitively to the number of constituents that must be added or removed in order to go from one k-test vector to the other.

```
k-test vector distance:
```

```
def distance(self, other):
    return len(self ^ other)
```

2.4 Tests

We make some tests to ensure that the implementation works at least superficially as intended:

```
tests = {
    'invalid example': ({'aa'}, {'aa'}, {'aaaa'}, {'ada'}),
    'aa+': ({'aa'}, {'aa'}, {'aaa'}, {'aa'}),
    'bb+': ({'bb'}, {'bb'}, {'bbb'}, {'bb'})
}
instanciations = {}

for name, parameters in tests.items():
    try:
        ktest = ktestable(*parameters)
        print('The creation of "%s" went well' % name)
        instanciations[name] = ktest
    except ValueError as e:
        print('The creation of "%s" failed:\n - ' % name, e)


union = instanciations['aa+'] | instanciations['bb+']
intersection = instanciations['aa+'] & instanciations['bb+']
symmetric_difference = instanciations['aa+'] ^ instanciations['bb+']

print(union.prefixes)
print(intersection.prefixes)
print(symmetric_difference.prefixes)

print(union.distance(union))
print(union.distance(intersection))
print(len(union), len(intersection))
```

```
The creation of "invalid example" failed:
 -  Incorrect prefix length, incorrect suffix length, short strings conditions not satisfied.
The creation of "aa+" went well
The creation of "bb+" went well
{'bb', 'aa'}
set()
{'bb', 'aa'}
0
6
6 0
```

3 Efficient algorithm

3.1 Union consistency definition

Before learning the union of languages, we need to ensure the union consistency between two k-test vectors Z and Z', *i.e.* the fact that the union of their languages should be the languages and their union. Linard *et al.*'s proposition 4 provides a way to do this.

Proposition 4 relies on padded prefixes and suffixes. A padded prefix is a prefix with an out-of-alphabet character $\bullet$ added at the beginning of its string. A padded suffix adds this character at the end of its string.

The idea is to create an oriented graph from the two k-test vectors, where a path starting from a prefix, ending in a suffix and passing through infixes will represent a word generated by the union of those k-test vectors. We will call this graph the consistency graph, and there are three aspects to it:

The vertices are the padded prefixes, the padded suffixes and the infixes:

$$V = \{\bullet u | u \in I \cup I'\} \cup \{u \bullet | u \in F \cup F'\} \cup T \cup T'$$

The edges are drawn from one vertex to the other if the suffix of size $k-1$ of the first vertex is equal to the prefix of size $k-1$ of the second vertex:

$$E = \{(au, ub) \in V \times V | a, b \in \Sigma \cup \{\bullet\}, u \in \Sigma^{k-1}\}$$

The colors are reflecting wether a vertex is "endemic" to one vector:

- a red vertex is endemic to Z,
- a blue vertex is endemic to Z', and
- a white vertex is endemic to none.

A vertex v is endemic to a vector $X = \langle I, F, T, C\rangle$ compared to another vector $X' = \langle I', F', T', C'\rangle$ if it appears only in X. More formally, it is endemic if the following holds:

$$\begin{cases} u \in I \setminus I' & \text{if } v = \bullet u \\ u \in F \setminus F' & \text{if } v = u\bullet \\ v \in T \setminus T' & \text{otherwise} \end{cases}$$

The paper shows that the union consistency is ensured if and only if there exists no path between a red vertex and a blue vertex. A path between red and blue vertices means that a word out of both languages emerges in the union, which is precisely what we want to avoid.

We will first compute the consistency graph and then test the union consistency. Both of these operations will be implemented into their own method of `ktestable`.

3.2 Consistency graph

For brevity and sanity's sake, we will use a library to do operations on graphs. We choose NetworkX[1] since it has proved to be easier to install than the alternatives.

We suppose NetworkX has already been imported like so:

```
import networkx as nx
```

The method `consistency_graph` returns both the graph and the red and blue nodes. It consists of three parts ;

[1] See `https://networkx.github.io/documentation/stable/install.html`.

Construct consistency graph:

```
def consistency_graph(self, other):
    <<Vertices construction>>

    <<Edges construction>>

    <<Graph assembling>>
```

We construct both the colors and the vertices at the same time since there is no reason not to. Rather than using padded prefixes and suffixes, we add a one letter prefix to each vertex.

Vertices construction:

```
def color_construction(left, right, nature):
    reds = left - right
    blues = right - left
    whites = left & right
    def add_prefix(set_):
        return {nature + el for el in set_}
    return (add_prefix(reds), add_prefix(blues), add_prefix(whites))

colored_prefixes = color_construction(self.prefixes, other.prefixes, 'P')
colored_suffixes = color_construction(self.suffixes, other.suffixes, 'S')
colored_infixes = color_construction(self.infixes, other.infixes, 'I')
prefixes, suffixes, infixes = map(lambda x: x[0] | x[1] | x[2],
                                  (colored_prefixes, colored_suffixes, colored_infixes))
reds, blues, _ = map(lambda x: x[0] | x[1] | x[2],
                     zip(colored_prefixes, colored_suffixes, colored_infixes))
reds = set(reds); blues = set(blues)
```

There are only three ways in which an edge can form between two vertices:

- a prefix can connect to an infix,
- an infix can connect to another infix, and
- an infix can connect to a suffix.

Edges construction:

```
edges = {(pre, inf) for pre in prefixes for inf in infixes
         if pre[1:] == inf[1:-1]}
edges.update({(left, right) for left in infixes for right in infixes
              if left[2:] == right[1:-1]})
edges.update({(inf, suf) for inf in infixes for suf in suffixes
              if inf[2:] == suf[1:]})
```

Since we are only interested by the paths between vertices, we construct the graph from the edges only, thus leaving out isolated vertices. In any case, there should not be isolated vertices because the k-test vectors are supposed to be well-constructed.

Graph assembling:

```
graph = nx.DiGraph()
graph.add_edges_from(edges)
return graph, reds, blues
```

3.3 Union consistency implementation

Union consistency:

```
def is_union_consistent_with(self, other):
    <<Path research>>

    <<Paths analysis>>
```

Searching for a path between reds and blues is akin to find a transitive closure and examine the reachability of red and blue nodes with respect to one another.

Path research:

```
graph, reds, blues = self.consistency_graph(other)
closure = nx.algorithms.dag.transitive_closure(graph)
red_reachable = {neighbour for red in reds for neighbour in closure.adj[red]}
blue_reachable = {neighbour for blue in blues for neighbour in closure.adj[blue]}
```

Finally, we only have to check if red vertices are reachable to blue vertices and vice versa.

Paths analysis:

```
if red_reachable.isdisjoint(blues) and blue_reachable.isdisjoint(reds):
    return True
return False
```

3.4 Tests

Some basic tests based on examples from the paper.

Union consistency test:

```
examples = {
    'z3': ({'ab'}, {'bc'}, {'abc', 'bca', 'cab'}, {}),
    'z4': ({'cb'}, {'ba'}, {'cba', 'bac', 'acb'}, {}),
    'z5': ({'ab'}, {'ba'}, {'abb', 'bbb', 'bba'}, {}),
    'z7': ({'ab'}, {'ba'}, {'abb', 'bbb', 'bba'}, {}),
}

instances = {iden: ktestable(*params) for iden, params in examples.items()}

print(instances['z5'].is_union_consistent_with(instances['z7']))
print(instances['z3'].is_union_consistent_with(instances['z4']))
print(instances['z3'].is_union_consistent_with(instances['z7']))
```

```
True
True
False
```

4 Sources

1. Linard, A., de la Higuera C., Vaandrager F.:Learning Unions of k-Testable Languages, (2018): `https://arxiv.org/abs/1812.08269`